

A Pattern Hierarchy Approach to Object-Oriented Systems Design

Srinivas Krishnan
Department of Computer Science
SUNY Brockport
Brockport, NY 14420

Faculty Advisors: Dr Sandeep Mitra and Thomas A. Bullinger¹

¹ArchSynergy, Ltd.
2500 Turk Hill Road
Victor, NY 14564

Abstract

Modern software development has focused on Agile Methodologies and code reuse based on Design Patterns for Object-Oriented systems. These methodologies have resulted in highly iterative, phase-oriented processes for software development (e.g., eXtreme Programming (XP) and Rational Unified Process (RUP)). However, systematic approaches for *traceability* of the development model as it is mapped from one phase to another are not well understood. This paper addresses this problem by proposing an approach based on *Pattern Hierarchy*. Design Patterns are well understood, but they historically apply only to the design phase. This paper proposes a pattern hierarchy in which a pattern is employed in each phase based on the appropriate level of abstraction. Initial studies with different processes suggest the following structure: Product Metaphor -> Analysis Patterns -> Design Patterns -> Usage Patterns/Coding Idioms. The fundamental hypothesis presented by this paper is that while there are different patterns applicable at each phase, these patterns are traceable throughout – i.e., the pattern at a later phase may be derived from the pattern applicable to an earlier phase. The paper focuses on a set of concepts that facilitate such traceability from one phase to the next. This hypothesis was tested using case studies for various systems (a rental system and an online collaboration tool) developed using a new process - the Software Engineering Effectiveness Model (SEEM™). The results demonstrate the efficacy of this approach in maintaining the architectural integrity of systems throughout the development process, resulting in highly maintainable and extensible implementations.

Keywords: Patterns, Systems Analysis, Design, Methodology, Hierarchy.

1. Introduction

There is a serious crisis in the software industry with the challenge to develop robust software that meets customer needs and conforms to strict quality control standards. Object-Oriented (OO) programming was intended to be the “big leap” to solve these problems, giving the programmer concepts such as encapsulation and inheritance, as well as “frameworks” employ at implementation time. Frameworks would then simply be “tweaked” with the details that were specific to a given project. Moreover, individual frameworks would neatly interface to one another. Reusability was thus an important benefit of OO systems design. This reusability aspect of Object-Oriented Programming (OOP) was captured by patterns – i.e., blueprints for design [1, 2]. The use of patterns was actually invented in the context of building architecture by Christopher Alexander [1, 2]. He was an architect who observed that buildings can be classified under types and their blueprints could be used to build similar buildings. For example, while building a house, the bedrooms are placed above on the second floor where it is quieter and the bathrooms away from the kitchen; this is a pattern that is used in multiple homes. The problem with building patterns is that they are rigid, and cannot be changed once committed to implementation. Software developers realized the benefits of using patterns and applied it with the added advantage of flexibility. Building Patterns are applied to the whole process

from blueprints to the implementation. However, in the software development domain, patterns are usually thought to be restricted to the design phase, where they are known as Design Patterns [4,9]. Software Engineers have expanded on the definition of patterns as “a core solution for a problem that occurs over and over again”. Such problems, however, occur throughout the development process, not just in the design phase, but even earlier. Consequently, we can think of software systems as entities that can be classified under a specific “taxonomy”, which indicates that these systems have essentially the same behavior. We can therefore conceive of “Analysis Patterns” [7] for a particular system. Unfortunately, even though Analysis Patterns may be created, they seem to have no relation whatsoever to Design Patterns for that system – at least as described in the existing literature [4,7,9]. The research conducted indicates that the disjoint nature of Analysis, Design and other patterns has proven to be a serious roadblock for Agile Methodologies [6] and OO development.

Agile methodologies like eXtreme Programming (XP) [3], Rational Unified Process (RUP) [10], etc. propose different flavors of iterative software development. Each iteration corresponds to the creation of a set of artifacts during various phases and a dated release of the product. This process allows modifications and additions at any of the various releases created before the product ships. The key to the success of these methodologies is the *flow* from one phase to another - each artifact in a particular phase should derive from one in an earlier phase. This flow keeps the design traceable throughout the process, and allows designers at each iteration to easily go back and trace the chain of artifacts they want to change as they incorporate new features for the next release. However, current Agile methods, even though they advocate iteration, do not pay attention to this inherent traceability amongst the artifacts corresponding to the various phases. This paper presents an approach to maintaining such traceability and is fundamental in ensuring proper iteration according to our current research.

SEEM [5] states that patterns are inherent in the entire development process. Patterns are "time-trusted" solutions to a problem – solutions that have already been well codified and thoroughly tested. There is no reason for patterns to be considered starting only at the design phase – in fact, as SEEM advocates, each phase should consider the appropriate pattern for that phase (starting from the Analysis phase). How the patterns from one phase flow to the next is the principal focus of this paper.

2. Pattern Hierarchy

Any software design process can be broken into the following phases - Analysis, Design and Implementation. The following patterns apply to each of these phases:

Phase	Patterns	Artifacts
Business Case Analysis	Product Metaphor	User Stories
System Analysis	Analysis Patterns	Analysis Sequence Diagram, Class Diagrams, CRC cards
Design	Design Patterns	Design Sequence Diagram, Class Diagrams, CRC cards
Implementation	Usage Patterns	Program code
Implementation	Coding Idioms	Program Code




Figure 1: Pattern Hierarchy

The behavior embodied by patterns in a particular phase maps to the behavior described by the patterns in the succeeding phase. Patterns appropriate for later phases may have a larger set of behaviors associated with them – however, each behavior in the latter phase should map to a behavior the earlier phase requires. Thus, each Product Metaphor maps to a set of Analysis Patterns. Similarly, each Analysis Pattern maps to a set of Design Patterns. When Design Patterns are realized in code, the manner of code usage (e.g., the use of a particular Application Programming Interface (API)) illustrates how the Design Pattern maps on to Usage Patterns/Coding Idioms. It should be noted that Product Metaphors and Analysis Patterns are problem-domain patterns, whereas Design Patterns, Usage Patterns and Coding Idioms are solution-domain patterns. Once a system has been created, tested and successfully deployed, the sequence of mappings leading to the implementation becomes a pattern that can be reproduced for similar systems. If a similar system requires implementation with a different technology (e.g., the use of C# and Microsoft’s .NET framework instead of Java and Java-2 Enterprise Edition toolkit (J2EE)), the pattern hierarchy may be directly applicable, and the only difference will be that new Usage Patterns and Coding Idioms (e.g., those appropriate for C#) will be used.

The concept of a pattern hierarchy does not change the activities required for each software development phase, nor the basic nature of the artifacts produced in each phase. However, it does set some guidelines on how to proceed with the activities in each phase. The highest-level pattern is the Product Metaphor. Each development effort starts by identifying a metaphor appropriate for that product. This metaphor is the ultimate abstraction for the software – e.g., if the development team has to create a product that manages the rental/return business workflow for a video rental store, and then the metaphor to use for this product is a "public library". The concept of a public library brings to mind a whole set of behaviors which both the customer and the development team understand well. Definition of the system to be implemented can therefore proceed nicely, since both the customer and developers are on the "same page" right from the start. Identifying a metaphor can be problematic but it should drive the whole process and hence every effort must be made to find the right metaphor. Developer experience will contribute to the ease with which this can be done.

Once the metaphor has been identified, the User Stories [3] (similar to the Unified Modeling Language (UML) Use Cases [8]) can be created on the basis of the metaphor. User Stories are “verbs” – each User Story describes one system feature that accomplishes a definite result for the user. For example, for a public library metaphor, some of the User Stories identified are: Rent, Return, Add to Inventory and Modify Inventory Item. Details about the User Stories should be written up as though they are completely in the *problem domain* – i.e., no solution level artifacts (such as the features of the programming language used) should be considered. Figure 1 represents a typical user story for the domain.

<p>Title: Rent Item (Book)</p> <p>Description: Rent an Item (i.e., a book) to a customer</p> <p>Precondition:</p> <ol style="list-style-type: none"> 1. Customer has account with the library. 2. Book has id (e.g., barcode) encoded on it. 3. Customer is physically present with his ID card and with the book at the counter. <p>Flow of Events:</p> <ol style="list-style-type: none"> 1. Clerk uses customer ID to retrieve customer record. 2. Clerk uses book ID to retrieve book record and get the rental price and rental duration from this record. 3. Clerk creates an Invoice for the customer including the book title, rental cost and rental duration on it. 4. Customer agrees to rental terms and pays the rental charge. 5. Clerk hands invoice copy to customer. 6. Clerk records the rental for the book, including the due date in it. <p>Results:</p> <ul style="list-style-type: none"> ➤ Book is successfully rented. 	<table border="1"> <tr><td>Product Metaphor</td></tr> <tr><td>Analysis Patterns</td></tr> <tr><td>Design Patterns</td></tr> <tr><td>Usage Patterns</td></tr> <tr><td>Coding Idioms</td></tr> </table>	Product Metaphor	Analysis Patterns	Design Patterns	Usage Patterns	Coding Idioms
Product Metaphor						
Analysis Patterns						
Design Patterns						
Usage Patterns						
Coding Idioms						

Figure 2: Rent Item (Book) User Story

Having identified the User Stories, the next step is to build an Analysis Model. When building the Analysis Model, the development team needs to consider *Analysis Patterns*. For systems consistent with the public library metaphor, we define Analysis Patterns as *workflow descriptions*. Each workflow description corresponds to the Flow of Events of a particular user story, and the description should precisely indicate the sequence of events, as well the system entities involved in the workflow (refer Figure 2 below). Identification of the appropriate system entities that play the right roles and assume the right responsibilities for handling the workflow is the principal objective of the Analysis phase. UML can be used extensively at this stage - the Analysis Model is typically comprised of UML Sequence Diagrams/ Collaboration Diagrams. Class Diagrams and CRC cards are then derived from the sequence diagrams. Identification of the right system entities, and assigning them the right roles and responsibilities, is a difficult task. The team can look at previous projects that were well documented and get some ideas from them. The roles and responsibilities of the entities identified at this stage must be consistent with good software engineering principles, and have *high cohesion* and *low coupling*. This can be achieved if we consider each

entity as a natural abstraction of something in the real world. For example, for the Rent Item user story in Figure 1, the workflow involves the following entities: *customer*, *rental item* (i.e., *book*), *rental invoice* and *rental record*. These are naturally present in the real world, and since we are modeling them appropriately in our system, we will have high cohesion and low coupling amongst them.

From the Analysis Model, the development process leads to the Design Model. Therefore, the next set of patterns to consider is Design Patterns - these patterns indicate the first step towards the use of solutions that can ultimately realize the behavior(s) of the User Stories in code. Designers use the analysis level artifacts to analyze and determine where design patterns can be applied to their design. This can be a cumbersome task if cohesion and coupling issues are not dealt with properly at the Analysis Model level. Good cohesion at the analysis level, for example, would show in the workflow a uniform manner of creating entities, irrespective of the way the actual entity itself is eventually implemented. This indicates the need for a creational design pattern, such as the Factory Pattern, which would identify a specific factory object that creates entities implementing a particular interface. Use of such a factory object would greatly enhance the extensibility and maintainability of the code. Similarly, the workflow at the Analysis level could show that an external actor is interacting with a set of identified entities. For example, in the “Rent Item” User Story shown in Figure 1, the developer would see that the workflow shown in Figure 2 indicates that the external actor (Clerk) is interacting with the *customer* and *rental invoice* entities. If this external actor were a human user, then the developer would conclude that this is a perfect case for the Model-View-Controller (MVC) pattern ([9]).

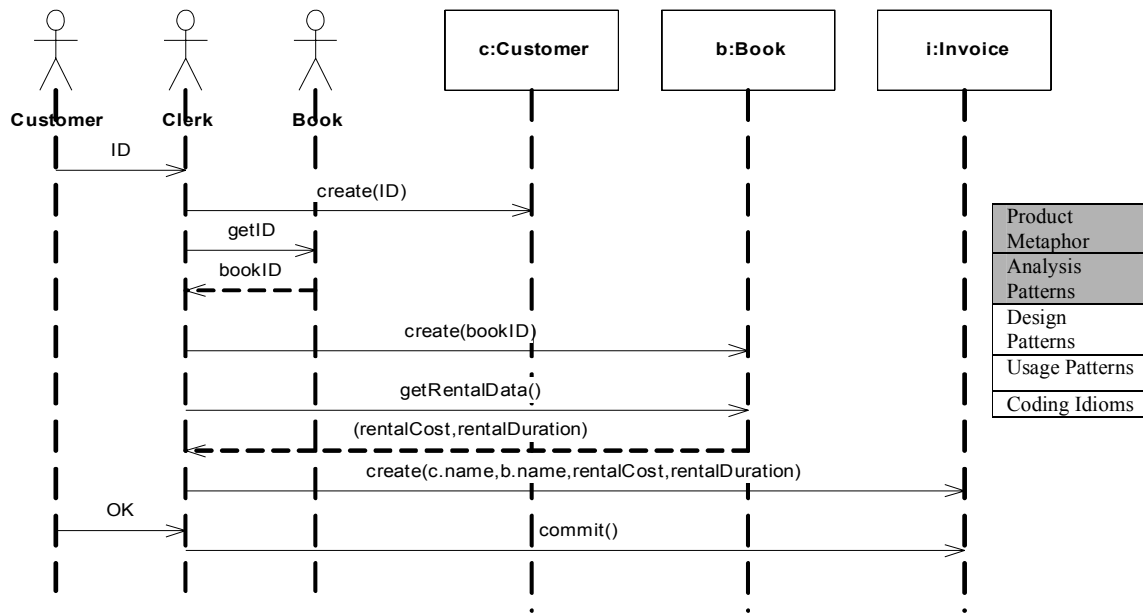


Figure 3: Analysis Sequence Diagram for Rent Book User Story

Given a particular workflow shown in an Analysis Model, it may not be immediately obvious which Design Patterns can be employed to move that workflow to the solution phase. However, if the workflow provides suitable abstractions, it is possible to identify the right design pattern. An example is the need for the use of a creational pattern (such as the Factory Pattern) while mapping the workflow shown in Figure 2 to the Design phase. The workflow does not explicitly spell out the need for a Factory pattern – however, the workflow is modeled in such a way that the use of a Creational Pattern to implement an aspect of the workflow becomes obvious after some thought. The workflow indicates the creation of a book entity as part of the rental business logic. The actual book created could be a soft-cover or a hardcover book, and the cost of the rental would depend on the actual book created. However, the workflow remains the same, irrespective of whether a soft-cover or hardcover book is being rented – namely, the cost of the rental has to be computed, the invoice issued and the customer balance updated depending on the money tendered. The workflow in the Analysis Model therefore deals with a certain abstraction – i.e., the rent-able book, with each of which a rental cost is associated. The Design Model, which flows from the Analysis Model, introduces a factory object in this workflow that can create a soft-cover or hardcover book, depending on the actual id of the book provided at run-time. The viability of a factory object at this point is

illustrated by the fact that we established good cohesion and coupling in the workflow embodied in the Analysis Model. The responsibilities of the rent-able book were clearly identified – all that the rent-able book had to do was know its rental cost and rental duration. Any real book – soft-cover or hardcover – can be assumed in the real world to “know” its cost. We therefore achieved good cohesion, which made it possible to introduce a factory object. If a different kind of book – say, a comic book – were introduced as a new feature in the public library, then it can be easily incorporated into the rental system. Only the factory object has to be made capable of creating comic books as well. The rental workflow does *not* change. Thanks to the proper, systematic mapping from the Analysis Model to the Design Model, the developers can clearly identify which parts of the system do not change, and which parts do. As described in the “Iteration” section below, following this process significantly enhances maintainability and extensibility.

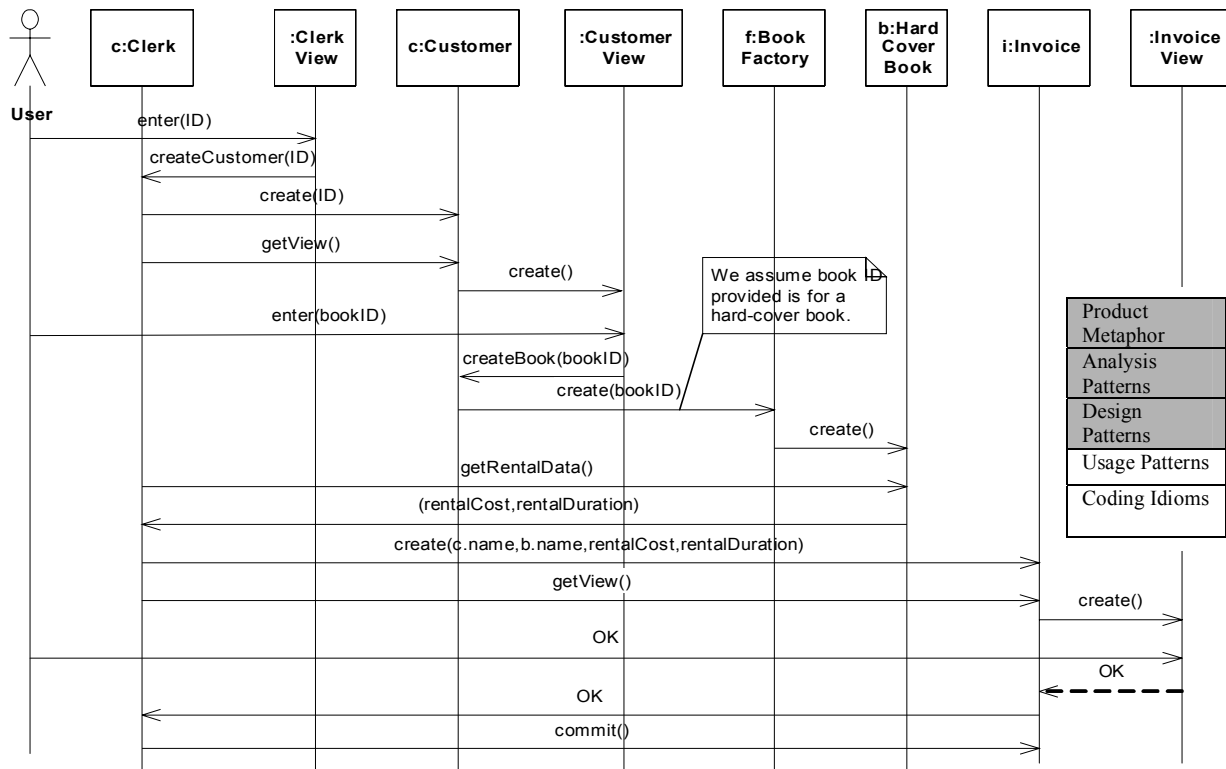


Figure 4: Design Sequence Diagram for “Rent Hard Cover Book”

The next phase of development is the creation of the Implementation Model from the Design Model – for software, this involves writing code. The translation from the Design Model to the code that constitutes the Implementation Model is achieved via the process of mapping each Design Patterns to associated *Usage Patterns*. Usage patterns are strictly solution domain patterns – each usage pattern is organized around a particular implementation technology “family” (e.g., Java/J2EE, C++/Microsoft Foundation Classes (MFC), C#/.NET, etc.). A usage pattern indicates the set of APIs and other features from the implementation domain that can be used to realize a particular behavior. For example, in Java it is possible to use the class “java.lang.Class” to create objects if we know the name of the class from which the object has to be instantiated. Thus, if we have a class in our rental system called *HardcoverBook*, then an instance of this class can be created using the following code (say, in class *BookFactory*):

```
public Book BookFactory(String bookType)
{
    /*
     * The values assumed by bookType are “HardcoverBook”, “SoftcoverBook”,
     * “ComicBook”, etc Each of these values should correspond to a class in the
     * implementation code.*/
    return Class.forName(bookType).newInstance(); }
}
```

Product Metaphor
Analysis Patterns
Design Patterns
Usage Patterns
Coding Idioms

This example illustrates one *possible* usage pattern for factory behavior realization in the Java technology domain. Of course, this usage pattern is not the only way in which factory behavior could be realized – numerous others are possible. Usage Patterns are many and varied – a family of network protocols including HyperText Transfer Protocol (HTTP) could be part of a usage pattern to realize a Web-based e-commerce site, a subset of the Java Database Connectivity (JDBC) API used to acquire pooled connections to databases, a set of locking primitives used to let “at-most-N” entities into a piece of shared memory – all of these are examples of usage patterns, and each of them are appropriate in their own technology domain. A new technology would mandate the creation of new, though similar, usage pattern.

The literature surveyed contains little work on usage patterns [4,7,9]. Most of the literature on a particular technology simply describes the features of the technology itself. For example, literature on J2EE describes the various kinds of Enterprise Java Beans (EJBs) available, how to set them up, configure them, etc. A usage pattern for J2EE, however, would relate the establishment and configuration of EJBs to a “problem situation” for which the EJB solution is appropriate. We envisage that these “problem situations” are actually Design Patterns. Thus, the usage pattern shows a way of mapping design patterns to code. If the available literature could present technology from this viewpoint, the benefits would be enormous. Productivity of code generation would be significantly enhanced, since there would be minimal “reinvention of the wheel”. Note that Design Patterns themselves are technology-independent. The Factory Pattern is the same, irrespective of whether it is realized in Java, Smalltalk, or C++. Today, if the developer needs to know how to implement a factory object in Java, then he/she reads a number of Java textbooks and looks up the Java API extensively. If for a later project, the developer has to implement in C++, then this whole process is repeated for C++. On the other hand, if there was a website where he/she could look up “Factory Design Pattern” and see its mappings to Java/C++/Smalltalk code, he/she would be significantly more productive. Of course, considerable work is involved in discovering and documenting these usage patterns. Also, for a particular situation, the usage pattern as documented may not be exactly appropriate and may need revision. However, we believe that the same arguments apply to Design Patterns. Therefore, efforts in discovering and documenting usage patterns would certainly be a worthwhile investment.

2.1 Iteration

The need for Iteration over the phases of a project is a “given”. In fact, Agile Methodologies like XP suggest that iteration be considered an integral part of the development process as functionality is added incrementally to the system over various releases. However, while methodologies like XP and RUP both advocate an iterative process, they offer few guidelines as to how to effectively execute a typical iteration. The concept of pattern hierarchies as outlined in this paper provides precise information as to how to complete a typical iteration. Let us assume that the need for an iteration has been recognized – and it is due to one of the “typical causes” – namely, either there has been a change in customer requirements or a defect has been indicated by testing. The first activity after such a need is recognized is to identify the “first point of impact” of the iteration. For example, a change in customer requirements would probably have its first point of impact in the User Stories, whereas an identified defect in the code would identify its first point of impact in one of the later phase artifacts – i.e., either the Analysis, Design or Implementation (code) artifacts. The defect could have arisen due to incorrect creation of any one of these phases. Let us assume that the project team realizes that the defect’s cause is due to an incorrect workflow in the Analysis model. The team then rectifies this error in the Analysis Model artifacts. Thanks to the strict traceability from the Analysis Model to the Design Model, the teams can now precisely and unambiguously determine how the Design Model will be impacted by the change in the Analysis Model. Once the Design Model is updated, the Implementation Model (code) can be updated in a similar manner. The new code, after testing, reveals the correction of the defect.

It is important to recognize that without such traceability across phase artifacts the tendency of project teams is to simply “hack” the code in an attempt to fix the defect. While it is possible that the team developed the code initially with good architectural concepts in mind and following a systematic process, it is highly likely (especially in order to meet deadlines) that iterations are dealt with only at the code level, and thus run the risk of breaking the architecture. By having the various phase artifacts and the nature of the traceability across the artifacts documented well, the probability of hacking is reduced, and iteration is much more systematic. The key is in the correct identification of the first point of impact of the iteration. Once this identification and modification is completed, following through on this modification to the code is systematic, and therefore, easy. As a result of this, the probability of achieving success in defect rectification is much higher than in following a “hacking” approach. Discovering, strictly following and documenting a pattern hierarchy that provides traceability across the various phase artifacts is therefore a worthwhile investment.

3. Conclusions and Future Research

The concept of patterns in software development has proven useful to many programmers, allowing them to reuse ideas that worked in earlier solutions, thus increasing their productivity and generating more reliable and maintainable software. This paper suggests that patterns not be confined to the Design phase alone. The basic philosophy of patterns applies to all phases of the development process, from Inception through to Implementation. The product metaphor that provides the basis for understanding new product to be developed flows through to the Analysis model, in which workflow patterns are used. If the workflow is modeled correctly at the Analysis level, then it can be mapped further to other Design level patterns, which can then map on to Usage patterns that can be used to correctly write the code conforming to the Design patterns. It is important to note that the patterns conceived of at each stage are documented properly in the artifacts appropriate for that phase, and the mapping from the patterns in one phase to the next is precisely and unambiguously indicated. The nature of the pattern hierarchy that is used for a particular project is therefore common knowledge for the whole team. When this is accomplished successfully for a particular team working on a particular project, then handling change is relatively easier. As mentioned in this paper, the team handles change by identifying the starting point of impact for the change, and following through with the traceability to code from that starting point. This has the potential to significantly reduce code hacking. Architectural integrity of the code is thus preserved in the course of handling change, and the system does not become unwieldy as a result of the accommodating the changes.

Future work in this area is expected to consist of investigations and case studies that validate the hypothesis presented in this paper. We have investigated the use of pattern hierarchies for workflow-oriented systems like the lending library example presented in this paper. The applicability of these ideas to a different class of systems – e.g., systems containing a significant degree of concurrent entities – will be an interesting topic to study.

4. References

- [1] Alexander, Christopher, (1979). *Timeless Way of Building*. New York: Oxford University Press.
- [2] Alexander, Christopher, Ishikawa, Sara, Silverstein, Murray (1977). *A Pattern Language: Towns, Buildings Construction*. New York: Oxford University Press.
- [3] Beck, Kent (2000). *Extreme Programming Explained*. Reading, MA: Addison-Wesley.
- [4] Braude, Eric (2004). *Software Design: From Programming to Architecture*. Hoboken, NJ: John Wiley & Sons, Inc.
- [5] Bullinger, Tom, Mitra, Sandeep, (2003). Software Engineering Effectiveness Model. Retrieved February 8, 2004, from <http://www.archsynergy.com/seem.php>
- [6] Cockburn, Alistair (2001). *Agile Software Development*. Boston, MA: Addison-Wesley.
- [7] Fowler, Martin. (1997). *Analysis Patterns: Reusable Object Models*. Boston, MA : Addison Wesley.
- [8] Folwer, Martin (September 2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Boston, MA: Addison-Wesley.
- [9] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- [10] Kruchten, Philippe (2000). *The Rational Unified Process: An Introduction* (2nd ed.). Reading, MA: Addison-Wesley.